European Journal of Engineering Education
Vol. 33, Nos. 5–6, October–December 2008, 587–596

Taylor & Francis
Taylor & Francis Group

Simulation framework for teaching in modeling and simulation areas

Marisa Raquel De Giusti[a]*, Ariel Jorge Lira[b] and Gonzalo Luján Villarreal[c]

[a]*Comisión de Investigaciones Científicas de la Pcia. De Buenos Aires (CICPBA), Proyecto de Enlace de Bibliotecas (PrEBi), Universidad Nacional de La Plata, La Plata, Argentina;* [b]*Proyecto de Enlace de Bibliotecas (PrEBi), Universidad Nacional de La Plata, La Plata, Argentina;* [c]*Consejo Nacional de Investigaciones Científicas y Téchnical (CONICET), Argentina*

(*Final version received 20 October 2008*)

Simulation is the process of executing a model that describes a system with enough detail; this model has its entities, an internal state, some input and output variables and a list of processes bound to these variables. Teaching a simulation language such as general purpose simulation system (GPSS) is always a challenge, because of the way it executes the models and the abstraction level it can achieve, very different compared with most well-known programming languages. This article presents an open source simulation framework that implements a subset of entities of GPSS, which could help students to improve the understanding of this language. This tool also stores all entities of simulations in every single simulation time, which is very useful for debugging simulations, but also for getting a detailed history of all entities in the simulations, knowing exactly how they have behaved in every simulation time.

Keywords: discrete event simulation; storing runs; statistical analysis; simulation teaching

1. Introduction

Block programming languages offer an important abstraction level that allows the programmer to map objects from the real system to entities of the simulation in an almost transparent way, providing implicitly with each object a set of functions and facilities, which lets the programmer focus on designing the model and forget about implementation details (Naylor *et al.* 1966, Dunna *et al.* 2006). This is particularly useful when looking for quick solutions, since time savings achieved can be very significant and thus the required cost for modeling, simulating and experimenting with systems is considerably reduced.

Teaching in the area of simulations is always a challenge (Wildenberg 1981, Jones 1983), because this involves a huge amount of new concepts and complex methodologies; if we add a new block simulation programming language, which implies a new programming paradigm and a brand new way of thinking programs, the challenge is bigger yet, particularly if we are dealing with students who know programming very well but come from a background of procedural, object-oriented or functional programming languages and a different way of regarding models (Wankat and Oreovic 1993, Morgan and Jones 2001). General purpose simulation system (GPSS)

*Corresponding author. marisa.degiusti@sedici.unlp.edu.ar

ISSN 0304-3797 print/ISSN 1469-5898 online

DOI: 10.1080/03043790802568138
http://www.informaworld.com

programming language, on which this work is based, is not an exception to this rule (Minuteman Software 2008, WebGPSS 2008).

Comprehension by some of the students about the way transactions move along blocks in a simulation written in GPSS and the way they interact with all entities is particularly difficult, since the language performs all these operations in the background and the programmer just sees the results. In addition, all these results are given once the simulation has ended and are not always useful for students who want to understand what, when and how events happened during the simulation.

In this work we present an open simulation framework that allows, on one side, to know, modify and extend the code of all blocks and main objects of the model, and, on the other side, to store in a database all simulation objects in order to know how they have evolved over time and what has happened in each simulation time. This permits the students, for example, to pick any entity and know all changes it has suffered in all simulation times, which entities it has interacted with and what its life time inside the model was. In addition, they may select a range of times and see what entities existed at that moment.

2. Brief introduction to GPSS

Since the framework we are presenting here uses GPSS's model and entities, we consider that these concepts must be clearly defined before going on. This is not a GPSS tutorial, and concepts mentioned here are definitely not enough to learn this language; interested readers are encouraged to visit Minuteman Software's web site, which has a complete and detailed explanation of the language and every entity of the model.

GPSS is known as a block-oriented simulation language; this means that programmers use block sentences to describe entities, actions and operations. This short description offers three important clues about this language.

- Many entities exist: along any simulation, many entities are created, and some of them are also destroyed. Consequently, we have two types of entities: permanent (the ones that are created and never destroyed until the simulation ends) and temporal, also called transactions (the ones that are created and destroyed during the simulation).
- Entities interact with each other, which we call actions: some entities offer one or more services whereas other entities consume those services; entities have to synchronize their activities, share data or work together in order to perform an operation. The important concept here is that entities are not isolated in the model but they interact among them and this makes the simulation moves dynamically.
- Operations are performed: this point is quite obvious but it is important to make it clear. GPSS aims to simulate mathematical models of real-world systems; naturally, many mathematical and statistical operations are performed. The programmer may create its own functions and processes; besides which, the system has embedded lots of routines that help the programmer write complex functions.

Let us explain how all these concepts work together; in other words, let us see how it runs a simulation.

All blocks are written in a sequence of sentences (jumps may exist inside). Then, this code is interpreted and, if everything is fine, executed. During the interpretation, many permanent entities have been created, but others might be created during runtime.

While the simulation is running, lots of transactions (maybe hundreds or thousands) are created; each transaction *moves* through the block list, executing different actions according to the block

they are passing by and going to the next block in the list if possible. If not, the transaction is stopped in one (of many) queue in the system and waits until some condition is achieved to go on. The important part of this explanation is that only one active transaction exists at any time; even though there is an internal clock that indicates the time in the simulation and that during one clock time many transactions may have moved, there can actually be one single transaction as *active* during any *real* time.

There is a special entity, the transaction scheduler, which *decides* which is the next transaction to become active. This scheduler uses two queues of transactions: one for current transactions (transactions that have to become active during this clock time) and one for future transactions (they will become active in a known future clock time).

3. The model

For the development of this framework a subset of GPSS entities have been selected, which allow us to execute simple simulations but that require the execution model to be complete enough. Among these entities we find transaction, facility and, clearly, simulation.

The simulation entity is in charge of the execution of commands and the generation of transactions; it is also the task of this entity to invoke the transaction scheduler which must decide which transaction have to be the next active transaction, and to manage the simulation clock, detecting when it must update and making all required tasks in each clock change. This implies the addition of transaction scheduler and system clock entities inside the model. A spare entity that deals with references to entities resolution has also been added, either by name or identifier; this entity is able to locate any entity at any time anywhere in the simulation.

All entities form a horizontal composition cyclic graph, being the entity simulation as the root of this graph, from which all other nodes can be visited (Figure 1). The entity transaction scheduler is

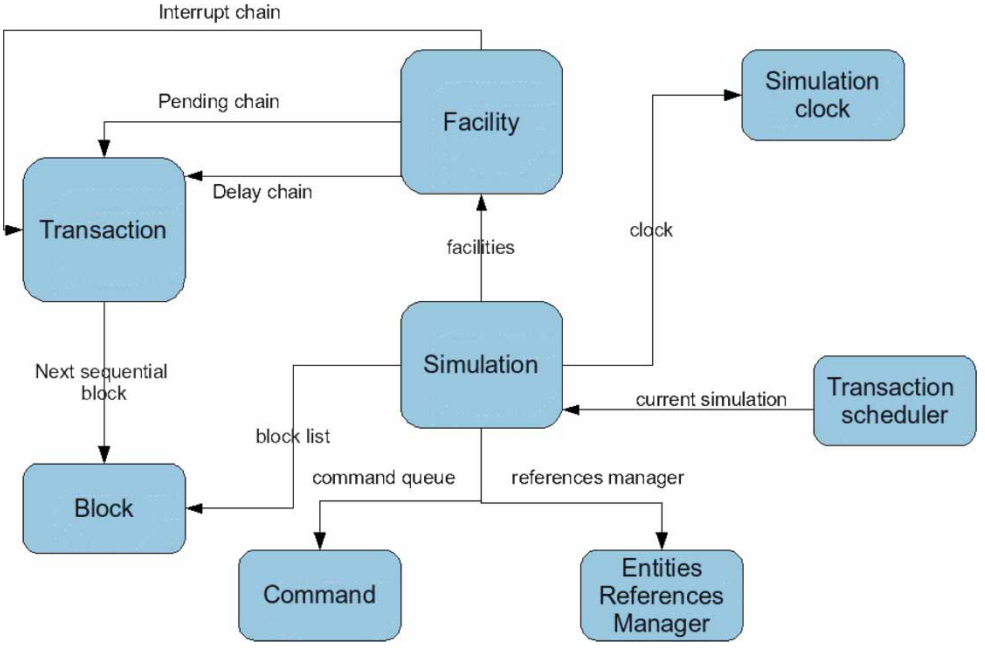

Figure 1. Entity composition graph.

extremely complex since it must interact, at first, with the two main chains of a simulation: current events chain and future events chain. But this entity must also be able to access transactions held by other chains in each existing entity every time is needed; for example, if the transaction is trying to release the facility, the transaction scheduler must select which transaction will be the following owner of the facility being released, meaning that it must analyse the current events chain, the delay chain of the facility, the preempt chain and the interrupt chain.

The movement among chains is also a difficult task, since it implies not only the exchange of hundreds or thousands of transactions from chain to chain, but it must also select transactions to exchange and choose the right moment to make this operation (change in the system clock, delay condition testing and others).

The decision to include the entity facility in this first version of the framework was not taken randomly; this entity involves a wide set of functions and entities, and has an intricate method for selecting transactions according to the internal state of each one of its chains and the way it is being accessed.

4. Simulation persistence

The aim of this framework is to provide an evolutionary analysis of the simulation, allowing access to the simulation state at any simulation time, once the simulation has ended. This means that all simulation-time instants have to be stored somewhere and retrieved when required, while the simulation is running and thus, the system clock moving. In this section, the storage and retrieval model of this framework is described, going from when objects are saved to how it is done.

In order to store a simulation in the disk, a relational database engine (MySQL) together with a java open relational mapping (Jordan & Russel 2003, ORM, Hibernate 2008, Java Data Objects 2008) named Java persistent objects 2008 (JPOX), has been used. The use of these tools lies on the fact that both are open source, easy to use, efficient and widely spread. However, the persistent module of this framework is very flexible, which means that the ORM can be easily changed (Hibernate, OJB, Versant), the database engine can be switched (Oracle, DB2, SQL Server, Postgress) or even the database model can be moved to an object-oriented database (such as Fast Objects), avoiding the use of an extra layer to map memory objects to rows in the database and vice versa. This flexibility permits programmers to adapt this implementation to their favourite flavors instead of forcing them to learn other tools of technologies. For example, if programmers are used to MySQL but prefer Hibernate, they can easily use it; or if they do not want to use MySQL and prefer Oracle, with OJB as the ORM, can perfectly adapt these frameworks.

As mentioned above, we are dealing with discrete simulations, hence we have a countable set of simulation instants in which many events have occurred and entities have been altered. In other words, any simulation has a time line with discrete, ordered and countable values, along which the simulation has existed and has had an internal state. If the simulation is thought like a connected graph of entities, as mentioned in previous section, the state of the simulation at any time instant is the state of the graph representing the simulation; and the state of the graph consists of the state of each of its entities and the connections among them. Keeping this concept in mind, we have considered taking a picture of the whole simulation, including all entities with their internal state, and storing this picture exactly as it is, with all entities, values and relationships inside the simulation graph. This way, every time the simulation clock is updated, just before these events actually occur, the storage process is launched; it is very important to make clear that this process starts before the simulation clock is updated, which ensures that the simulation is being stored with this clock value, before it changes to the following time instant.

There is a special case in the first simulation time, when there is no current state to store before the clock changes, but it is desirable to know how the simulation was at start time. This issue is solved in a very practical way: before the simulation actually starts, it is prepared (by GPSS commands and also by few framework constraints and warm-up functions) and then it starts running; so, this startup state launches the first storage process in zero clock time, and then the simulation runs as expected.

To store a simulation in a determined time t implies storing all entities that take part in this simulation at this time t. In order to collect all entities and values, the composition graph is walked along, starting from the node representing the simulation entity and moving along all simulation chains, entities list and chains of every entity. Since blocks are also entities, they must be persisted too.

Every time the graph is walked along, all information that can be persisted is collected and, once it is finished, the simulation clock is updated and the execution goes on. Then, ideally, all these data should be persisted to the disk and the system clock updated in order to go on with the run. The problem is that writing to the disk is too slow compared with main memory and processor speed, and the simulation cannot be stopped until objects are stored in every change of the system clock; to avoid this, the framework separates the data collection from delaying the object persistence, and makes use of concurrent programming to improve this process.

The solution to the delay problem consists of having a ready-to-store queue, and a low-priority thread in charge of the persistence. All data ready-to-persist are queued and remain there until it is actually persisted, and the simulation continues running without waiting until data are written to the disk. In order to make the actual persistence of the objects, a single thread has been included, which runs together with the rest of the simulation. This thread picks up simulations ready from the persist queue and deals with DAO object and all database items (connections, queries, transactions and so on). It is important to remark that the extra thread has a lower priority than the simulation run; we have designed it this way because we do not want the simulation to be delayed for any reason.

This solution is good for providing immediate results when the simulation ends while the persistence to the database is being made in the background. When the simulation run has ended, a report is printed, showing the final state of the permanent entities, all named values, the last clock value and some other useful information. This way, the programmer may start analysing data or even changing or adapting the model while the simulation is constantly stored in the background, improving user experience and efficiency. Then, when all data have been persisted, the user can start querying and exploring permanent entities as well as any transaction.

4.1. *Collection of data*

The whole application is written in Java, which means that it uses all well-known object-oriented concepts (inheritance, hierarchy, OID, etc; Barry 1996). When saving these objects to the database, we must make sure that they are not updated since we would lose the state of the objects before this update. One solution would be to have a version scheme where only new or changed data are stored; the main problem of using versions is that it would be really difficult to retrieve a simulation in any system clock. It would be efficient in the disk, but slow in data retrieval (Kim 1988, Kim 1990, Balsters *et al.* 2001).

Since hard disks are really huge these days, and computers have lots of memory, it is not expensive to duplicate data even if it has not changed. Having this idea in mind, the solution applied in this framework has been both simple and efficient: the whole simulation graph is cloned in every change of the system clock, copying every node that takes part in the simulation (transactions, facilities, blocks, chains and everything else). Once the cloning process has ended,

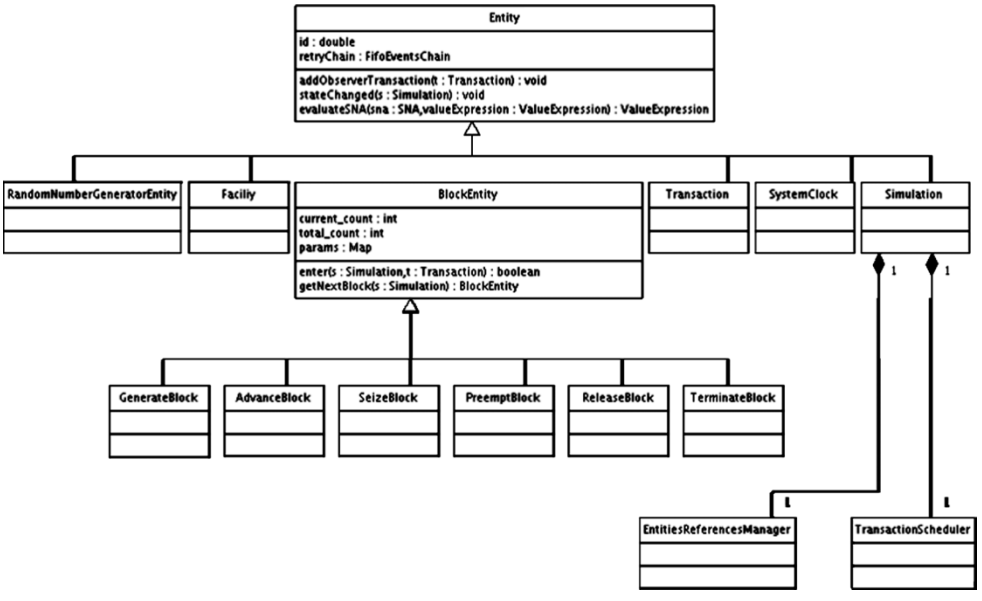

Figure 2. Simulation cloning process.

the cloned simulation is queued to store and, as noted above, the simulation can move on. This is illustrated in Figure 2.

In order to make this cloning process easy and exstensive, every entity implements a clone method, which performs two very important tasks (Figure 2).

(1) Creates a new clone object of itself, with all information that this object considers should be copied (not all information of all entities is stored in the database, since not everything is useful for further analysis). Before creating a copy, every entity must verify that there is no another copy of itself already in memory. This could happen since the graph is cyclic, thus any entity might be *visited* more than once. If this is the case, the object simply fetches and returns the copy already created and finishes the process. Otherwise, the original object creates a copy and registers a new pair <key,value> in a global dictionary, where the *key* is the original object, and the *value* is the cloned object.
(2) If the clone object has just been created, then the original object launches this cloning process to all objects reachable from it. This recursion ensures that the whole graph will eventually be cloned, and together with task 1, that all entities are cloned only once. Besides entity values, all entities references and connections are set in the cloned objects, which means that the graph structure is cloned too.

In order to make this cloning process as efficient as possible, the global dictionary has been implemented using a hash table, making searching and fetching objects a very fast task. This hash table is released after every clock update, in order to allow the process to take part in the following time-instant.

5. Data recovery

The way data are stored permits the easy retrieval of simulations; in the database there are many copies of every simulation run, where each copy will differ in, at least, the state of the simulation clock. Hence, to recover a simulation in a specific time t is as simple as retrieving the simulation

object and again, starts a chain effect retrieving all objects reachable from it. Again, trying to make it as efficient as possible, the retrieval of entities from the database is made once and kept in memory for further processes.

To have all this information well organized is crucial, since its purpose is to use it for analysis and, if it is not properly discriminated, it is very hard to take advantage of it. Because of this, the whole simulation run is stored in an increasing sorted list, where every list item consists of a system clock and the whole simulation graph in this clock time. This approach allows the retrieval of any simulation picture at any clock time, or to pick up any entity at any time and get to know what it looked like.

Being able to retrieve a simulation in a time t is not the only advantage of this tool. All entities, both permanent and temporary, have an internal identifier, which never changes along the simulation. This is very useful for writing new analysis tools that pick a specific entity and all copies generated before it; with all this information, this tool could show exactly how the entity changed, or which transactions were accessing it or were in its chains, or any other data that determines its internal state. This tool could even go beyond these ideas: we could have stored many simulation executions for the same model but with different parameters, and then we could retrieve any particular entity with all its states for all simulations. This way, we can not only see how this entity has changed, but also compare these changes with all instances of the same entity in different simulations.

6. Access to code and model extension

The framework proposed in this work is totally open. This allows access to the code that implements by each entity and each block, or even the extension or improvement of this code. This way, any student from the simulation's area can understand how each entity works inside or what exactly each block does every time the transactions invoke its routine. This permits the learning of new entities and blocks to be faster, or even to propose improvements, new functionalities or blocks, from the real knowledge of each one of them.

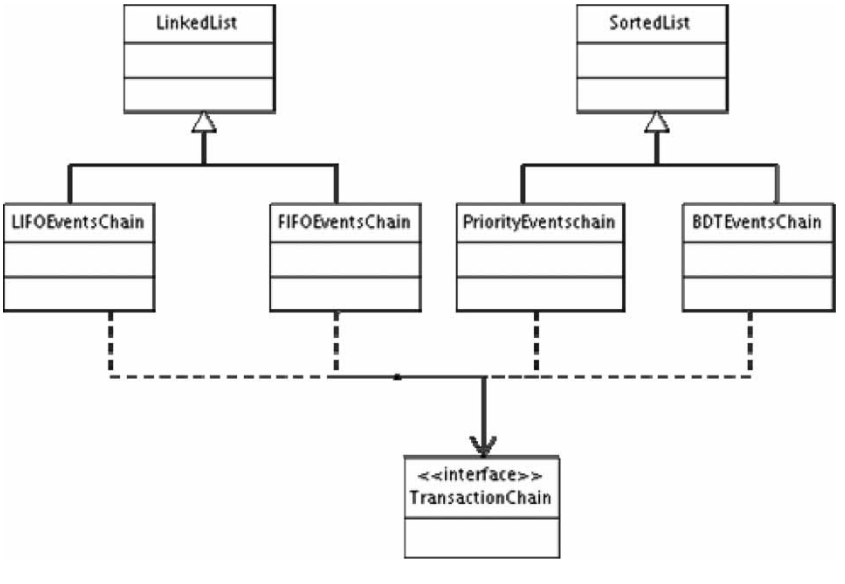

Figure 3. Entities hierarchy.

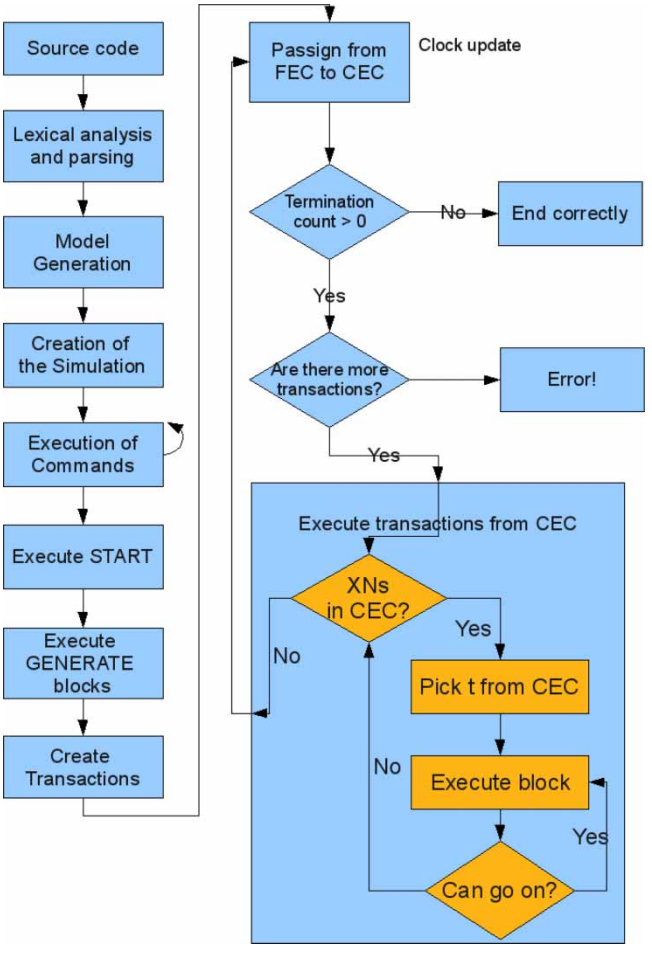

Figure 4. Chains hierarchy.

The design proposed here is thought to be easily extentable by any programmer, allowing the incorporation of new entities and block with very little coding. The entities have been conveniently disposed in a hierarchy (Figure 3), placing in the upper classes the state and the common functionality to every entity and leaving for the lower classes the specific behaviour of each of them. The blocks are part of this hierarchy/rank, because they are also GPSS organizations, but they possess a special/particular hierarchy because they have a slightly different behaviour due to their nature as executable entities. The existent model also includes the main chains, both the ordered ones from of the time they entered (first in first out (FIFO), last in first out (LIFO)) and as the ones ordered according to different transactions' parameters (priority, bdt). Again, the chains have been disposed of in a hierarchy (Figure 4), which allows incorporating new chains with different ordering mechanisms.

7. Reactions

After using this framework during a few classes in late 2007, in general the reactions from students seem to be mostly positive. Students are inclined to easily learn to write models and run them using this tool; most students have emphasized the similarity with Minuteman's Software GPSSw

editor and the speed with which the frameworks shows results even though data are still being stored. However, there have been some complaints about the tool:

- Lack of documentation: there are no user manuals or tutorial yet, and students need some sort of written support. The framework is easy to use in its most simple way, but it has hidden menu tools and options which are hard to use if there is no help at all.
- Lack of analysis tools: after the simulation ends, the report is displayed and students may access transactions, chains and facilities detailed lists; we know – as developers of the framework – that this is not enough, and students noticed this immediately. They claimed more and more analysis reports, navigation-through-the-graph tools, graphics and statistics. As a positive reaction, students have thought up and proposed new and different ways to show results and navigate among entities in the model; although they are being checked (both for usability and possibility of implementation), we consider it very positive that the students have created new ways of showing information about simulations.
- Model is not complete: we are aware of this and it has been explained to the class, but they wanted to see the whole model.

Besides the 'negative' reactions, there have been many very positive and unexpected reactions from some students. First, as we have mentioned above, they proposed many ways to make reports; second, when we said that the framework was open source – and that we have the source available – they asked for the source code and began to observe the implementation of the entities; the transaction scheduler seemed to attract all their attention. Lastly, after they saw the model and the code, they proposed changes, different implementations of existing entities and they started thinking of ways to implement other non-existing entities (such as groups, switches and storages); this was unexpected, we were surprised to see how easily they understood the source code and started improving it.

8. Conclusion

In order to help students during the learning process of GPSS (and other simulation tools), we have studied the GPSS model and grouped a subset of entities to develop a base model, which requires extension but is complete enough to start using simple yet powerful simulations.

Besides the completion with the rest of entities, blocks and commands, the improvement of the graphic interface should be an immediate improvement. Currently, these tools have few windows to display transactions and facilities entities, and there is a special window in which users can access all the information of a facility, including owner and transactions in all chains. There is a very useful extension in which users could use an iconic interface to program the models, using boxes and arrows representing entities and interactions among them (Dunna *et al.* 2006). The use of RCP technology (http://wiki.eclipse.org/index.php/Rich_Client_Platform, Eclipse 2008) permits us to include high-quality graphics that reflex some data from objects persisted in the database, and to integrate this framework with existing ones and with the ones being created. Additionally, programmers could start testing other DBMS (relational or object oriented) and analyse performance, speed and ease of use. The list of possible improvements and extensions could go on and on.

Although the aim of this framework is to help teaching and learning in simulation areas, there are many other obvious and not so obvious purposes for a development like this. As any other open source project, this framework lends itself to being extended and improved by people around the world, which would not only complete the model but also add quality and new analysis tools, as described above.

References

Balsters, H., de Brock, B., and Conrad, S., ed., 2001. *Database schema evolution and meta-modeling: 9th international workshop on foundations of models and languages for data and objects FoMLaDO/DEMM 2000 Dagstuhl*. 1st ed. Springer.

Barry, D.K., 1996. *The object database handbook: how to select, implement, and use object-oriented databases*. Wiley.

Dunna, E.G., Heriberto, G.R., and Leopoldo, E.C.B., 2006. *Simulación y Análisis de sistemas con ProModel*. London: Pearson.

Eclipse. http://www.eclipse.org

Hibernate ORM. http://www.hibernate.org/

Java Data Objects. http://java.sun.com/jdo/

Java Persistent Objects. http://www.jpox.org/

Jones, K., 1983. *Simulations language teaching (new directions in language teaching)*. Cambridge: Cambridge University Press.

Jordan, D. and Russell, C., 2003. *Java data objects*. Sebastopol: O'Reilly Media.

Kim, H.J., 1988. *Schema versions and DAG rearrangement views in object-oriented databases*. Technical Report. Department of Computer Sciences, University of Texas at Austin, USA.

Kim, W., 1990. *Introduction to object-oriented databases*. Cambridge, The MIT Press.

Langefors, B., 1985. *Teoría de los sistemas de información*. 2nd. ed. Buenos Aires: EL Ateneo.

Minuteman Software. http://www.minutemansoftware.com

Morgan, R. and Jones, K., 2001. The use of simulation software to enhance student understanding. IEEE, Vol. 2(5), pp. 331–336.

Naylor, T.H., *et al.*, 1966. *Computer simulation techniques*. 1st ed. Hoboken, New Jersey: Wiley.

Wankat, P.C. and Oreovic, F.S., 1993. *Teaching engineering*. New York: McGraw Hill.

WebGPSS web site. http://www.webgpss.com

Wildenberg, D., 1981. *Computer simulation in university teaching*. Amsterdam: Elsevier.

About the authors

Gonzalo Luján Villarreal is an Analyst in Computing (2005) and Licenciate in Systems (2008) at UNLP. He has worked on the PrEBi and SeDiCI projects since 2004 in the software modeling and development, and he has researched in areas related to digital image processing, digital libraries and computer simulation, the subject in which he focussed in his final thesis. Since 2005, he has been teaching at the Computer Science College in the subject Simulation and Models. In 2008, he has receive a posgraduate scolarship from CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas), and he is currently doing his PhD in Computer Science together with a master's in Software Engineering.

Ariel Jorge Lira is a Licenciate in Systems at UNLP (2008); he has worked at UNLP's PrEBi and SeDiCI projects since 2005 in software modeling, development and maintenance. He is currently leading the development of the Celsius Network, the last version of Celsius software. His researchs have been focussed on XML databases, especially very big databases, O/R mappings and OO databases, OO programming, design patterns and web technologies and techniques in software engineering and development.

Marisa Raquel De Giusti is an Engineer in Telecomunications at UNLP and a researcher in Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CIC-PBA). She teaches graduate and postgraduate courses at Computer Science College at UNLP, and since 1990 has worked with the Iberoamerican Science and Technology for Education Consortium (ISTEC). She is currently Director of Research and Development of ISTEC's Library Linkage project (LibLink). In 2008 she obtained the degree of Professor in Literature granted by UNLP. She initiated the PrEBi (Proyecto de Enlace de Bibliotecas) for UNLP in 1997, serving so far as its Director. In 2003 she has created the SeDiCI project, directed by her too. De Giusti has worked in many scientific areas and has authored more than 50 international publications, many of them related to statistics, experiments, simulation, information and digital libraries areas.